A Technique for Removing an Important Class of Trojan Horses from High Order Languages*

John McDermott

Center for Secure Information Technology†
Naval Research Laboratory, Code 5540
4555 Overlook Ave.
Washington, DC 20375
mcdermott@nrl-css.arpa‡

1 Introduction

In his 1984 Turing Award Lecture [4], Ken Thompson described a sophisticated Trojan horse attack on a compiler, one that is undetectable by any search of the compiler source code. The object of the compiler Trojan horse is to modify the semantics of the high order language in a way that breaks the security of a trusted system generated by the compiler.

The Trojan horse Thompson described is a form of virus, inasmuch as it is self-reproducing, but it has other characteristics that differentiate it from viruses that exploit the implementation details of a computer system. First of all, the self-reproduction is symbiotic, that is, the Trojan horse depends on the source text of the legitimate compiler for its continued existence. The virus only reproduces itself in the output stream of the compiler, when the compiler is compiling itself (thus destroying the original virus). A second difference is the relative portability of the virus to different systems. The compiler Trojan horse Thompson described is less dependent on the design details of a particular machine because it exploits the portability of high order languages. A final difference is the location of the virus in the executable file. The compiler Trojan horse is inserted in a place that is hard to search, that is, in mid-file. While this is possible for any form of virus, it is more difficult for viruses that do not have the compiler's functions at their disposal.

*Proc. 11th National Computer Security Conference, pages 114-117, October, 1988, Baltimore, USA, with typographical corrections.

†Author's current address: Center for High Assurance Computer Systems

‡Author's current email: *John.McDermott@NRL.Navy.mil*.

In his lecture, Thompson asserted that "no amount of source-level verification or scrutiny will protect you from using untrusted code." When no other means is used to provide assurance, this is true. However, this paper describes a technique that will remove such Trojan horses when used in conjunction with high-order language source code analysis.

This paper does not address what is referred to here as a *general virus*, that is, one that infects programs by direct modification of their images on disk (or other secondary storage). The general virus is a much larger problem; for example, detection of arbitrary general viruses is undecidable [1].

Section 2 of this paper explains why this class of Trojan horse and virus is important for trusted systems, Section 3 describes the basic compiler Trojan horse, and Section 4 describes the defense in detail. Section 5 gives a brief sketch of some measures and countermeasures. The paper concludes with some possible applications of this technique to building trusted systems.

2 The Importance of Compiler Trojan Horse Viruses

Compiler Trojan horses that are based on a symbiotic relationship between the source code and the binary code of the same compiler are important for several reasons besides the difficulty of detecting such a Trojan horse. Such a Trojan horse can compromise the security of many systems without propagating itself to those systems, it can compromise several classes of systems without redesign, and it is difficult to locate in an executable file.

The compiler Trojan horse can introduce unauthorized bypasses of security mechanisms into trusted systems, yet never exist on those systems. If such a Trojan horse is successfully installed on a development system, it can infect every system ever developed there. It can do this because the compiler will generate security flaws in every trusted system it generates code for, whether it is compiling for its own host or some other system.

Unlike the general virus, the compiler Trojan horse virus does not depend on low-level machine or operating system dependent details of the implementation it infects. One of the subtleties of Thompson's design is the use of the compiler's code generator to install the binary version of the viruses. If the Trojan horse virus is created in the same intermediate language that is passed to the code generator, it will be appropriately translated, linked, and loaded for whatever machine the compiler is targeted for. This applies not only to the self-reproducing or viral component, it also applies to the other malicious code that may be installed by the Trojan horse.

The compiler Trojan horse virus can easily install itself in the middle of the disk image of a program, because the compiler inserts it as it generates code. A general virus can be inserted into the middle of the disk image of a program too, but the problem is more difficult for the general virus. In the latter case, the virus must be designed to obtain privileges that permit it to modify the disk

images of programs. It must also be designed to perform link editing correctly, a non-trivial task when the program disk image is in link editor output format.

An assertion that general viruses cannot insert themselves in mid-program would be untrue. Nevertheless, it is significantly more difficult to create such a general virus and each copy of the general virus must contain all of the code necessary for correct mid-program insertion.

3 The Trojan Horse

The compiler Trojan horse is introduced into the system by a procedure that resembles a compiler compiling itself. The authorized Trojan-horse-free version of the compiler is used to compile a Trojan horse version of itself that always reproduces the Trojan horse in the compiler. For this to occur, there must be at least one read access to the legitimate compiler source code, and one or more execute accesses to the compiler (even viruses must be debugged), and at least one write access to the legitimate compiler binary. None of these accesses must necessarily occur on the same system.

The compiler Trojan horse is created as follows:

1. a source code version of the compiler is written that includes two Trojan horse capabilities, a security mechanism bypass and a self-reproducing feature,

2. the legitimate Trojan-horse-free version of the compiler is used to produce an object code version of the Trojan horse compiler, and

3. the Trojan horse object code version is installed over the legitimate object code version of the compiler.

At the end of this process, the source code of the compiler is free of Trojan horses but the executable file has two Trojan horse features. The first Trojan horse feature adds unauthorized security mechanism bypasses whenever it compiles the appropriate source code. The second Trojan horse feature reinserts the object code of the Trojan horse into the compiler whenever it it compiles the legitimate compiler source code. The Trojan horse code will remain in the system undetected by any analysis of the high order language source code of the compiler. Additionally, targeted security mechanisms generated by this compiler will have unauthorized and undetected bypasses installed.

This attack destroys the semantics of high order languages and undermines trust in assurance measures that depend on them. This threat is a serious problem because many valuable assurance techniques depend on high order language semantics. The availability of a technique for ensuring that the semantics of a high order language are free of such problems is essential for trust in high order language assurance techniques.

4 The Defensive Technique

The defense against this attack is based on the same concept of a compiler compiling itself. The defense exploits the symbiotic relationship between the source text of the legitimate compiler and the self-reproducing feature of the Trojan horse object code [3].

The Trojan horse reproduces itself whenever it compiles the compiler. To do this, it must first recognize some key portion(s) of the source text of the legitimate compiler. If the Trojan horse compiler compiles a compiler that it cannot identify as such, it will not reproduce in that program's object code. If the suspect compiler is fed a disguised version of itself, or a simple temporary compiler of a different design, it will not reproduce any such Trojan horses.

If the legitimate compiler can be hidden from its Trojan horse symbiont, so to speak, the Trojan horse will be erased by itself. Since it will not reproduce and the object code generated by the disguised compiler will be used in all future compilations, the threat is removed.

Hiding the legitimate compiler is not necessarily simple. First, recognition of an arbitrary compiler from an analysis (by the Trojan horse) of the function of the program can be ruled out as beyond the state of the art. However, the Trojan horse has many clues at its disposal, within the superficial features of the legitimate source text. If, for example, every identifier was scrambled, the Trojan horse still might detect a pattern in the keywords of the compiler source. If these were scrambled too (this is not difficult on many compilers), the Trojan horse could analyze comments, constant strings, such as "C Compiler, Version 3.7", requests for library units, or error messages (e.g. the C compiler is the only one with 198 error messages).

With careful analysis and design (by the defenders), a reduced function compiler could be created, one that even incorporated large amounts of code that had nothing to do with compilation. This temporary compiler would be compiled first, and then its Trojan-horse-free object code version used to compile the undisguised legitimate compiler. An integrated editor, compiler, and link editor that can scramble and unscramble program text as appropriate would be a very useful application of this technique.

5 Measures and Countermeasures

There are compiler Trojan horse measures beyond the text string searches mentioned above. Three measure that are of practical significance are, in increasing order of effectiveness for the Trojan horse:

1. attempt to recognize the compiler by its function,
2. make the Trojan horse part of the compiler's legitimate features,
3. extend the symbiosis.

The following discussion explains each measure and proposes a countermeasure to meet it.

5.1 Recognizing the Function of the Legitimate Compiler

While the problem of recognizing the function of an arbitrary program is not decidable, it is possible for the Trojan horse to recognize the function or a key part of the function of a restricted class of compilers. By making simplifying assumptions, the Trojan horse can still identify the legitimate compiler. An example of such an assumption is to assume the compiler will be generated by compiler construction tools. The output of a lexical analyzer generator or a compiler compiler contains regular patterns of instructions and data that could be associated with the tool and its target program.

The countermeasure to this approach is to construct a very simple but extremely modular compiler by hand. This compiler could employ techniques not used in compiler construction tools, to complicate the detection problem. If this compiler is also divided into very small modules in separately compiled source files, the Trojan horse will never have enough source text to identify the compiler function.

5.2 Installing the Trojan Horse in a Compiler Feature

One measure mentioned in [3] is to incorporate part of the compiler function into the Trojan horse. When the Trojan horse is removed by recompilation, no compiler on the system will work. A good concrete example of this is run-time support for the language. If the Trojan horse is placed in the object code of the run-time support mechanism of the language, in such a way that it performs some of the run-time support, it cannot be removed without breaking the language.

Any practical plan for disguising a compiler must take a system view of the entire language. Many languages assume the existence of support mechanisms that are outside the definition of the language or are not part of the compiler program. These support mechanisms can be suitable targets for the compiler Trojan horse, so this defensive technique should be applied to them also.

5.3 Extending the Symbiosis of the Trojan Horse

The final and most effective measure for the Trojan horse is to extend the symbiosis. As originally defined, the Trojan horse is a symbiotic system where the legitimate compiler source text is necessary for the continued existence of the object code virus. If a general virus is added as an additional symbiotic feature, the total Trojan horse system becomes a general virus with limited compiler Trojan horse capabilities, since the security mechanism bypass feature is still portable.

In the original case, if the object code of the compiler was modified without including the symbiotic self-reproducing feature, the compiler Trojan horse

would be destroyed when the compiler was recompiled. However, in the original case, the compiler Trojan horse with the self-reproducing feature is protected by its relationship to the compiler's legitimate source; whenever the Trojan horse detects the compiler in its own input, it reproduces.

In the case of the third measure, extending the symbiosis, the compiler Trojan horse system is expanded to include a general virus that resides at the start of some other program. That general virus component can then protect the original compiler Trojan horse. To do this, the general virus modifies the source of the compiler to include the compiler Trojan horse, recompiles it, and deletes the modified source file. All of this can be done in background mode, in such a way that the original compiler source is not modified, only a copy of it. The Trojan horse in the compiler can check selected executable files to see if its symbiont general virus is there, and if not, restore it. Thus both virus and compiler Trojan horse would exist in mutually reinforcing positions on an infected system.

The third measure results in a general virus, that is, the resulting Trojan horse is no longer strictly in the class addressed by this technique. The offending program gives up portability and relative simplicity for an increased likelihood of survival on a single hardware base. It increases its chances of survival by establishing a symbiont that will not be overwritten when the compiler is recompiled.

Nevertheless, the defensive technique of disguising the legitimate complier will present the symbiotic general virus with the same problem. The general virus must also identify the source text by the same means as its partner within the compiler object code. If the compiler has been successfully disguised, it will be equally protected from both the original Trojan horse and any symbiotic extensions that must identify the compiler. However, the disguise techniques must now be in effect at all times, and furthermore, all source files must be disguised in addition to the compiler.

6 Conclusions

In general, this technique increases the complexity of the problem facing a compiler Trojan horse. A more complex compiler Trojan horse is more likely is more likely to have errors, it will take longer to design, develop, and test, and it will probably be easier to detect because it is larger. Use of this technique makes the outcome of such an attack depend more on personal skill and less on system features.

The technique can be applied with varying amounts of resources and achieve reasonably proportionate assurance results. A low resource approach would merely scramble identifiers and constants before recompiling the compiler, while a high resource approach would be a programming system that completely hid all of its data. In the latter case, vendors could have such systems independently verified.

This technique is appropriate for very high assurance systems (e.g. beyond

A1) where every available defensive measure is desired. A high assurance version of this technique should be supported by a trusted development environment and additional measures to cope with general viruses [2, 5].

The technique is also appropriate for some systems of moderate assurance, depending on their design. Moderate assurance systems may contain components that are very vulnerable to Trojan horse attacks. Consideration should be given to using a low-resource version of this technique to provide some assurance that the development system has not introduced a Trojan horse.

A final practical issue is the examination of the high order language source for the compiler. The technique described in this paper assumes there is no Trojan horse in the source text of the compiler. This assumption may be difficult to meet for two reasons. First, the source code for the compiler is usually very closely held by the compiler vendor. This raises the question of certification and publicly available source text for temporary filter compilers; both are beyond the scope of this paper. Second, contrary to the statement in [4], the task of insuring that the compiler source code is trustworthy is non-trivial. Examination of compiler source code is probably the first use that should be made of practical automated high order language analysis techniques. Since the compiler source code is the most sensitive source code associated with the creation of a trusted system, it should be the most profitable place to look for high order language security flaws.

References

[1] F. Cohen. Computer viruses: theory and experiments. In *Proc. 7th National Computer Security Conference*, pages 240–263, September 1984.

[2] F. Cohen. A cryptographic checksum for integrity protection. *Computers & Security*, 6(6):505–510, Dec. 1987.

[3] S. Draper. Trojan horses and trusty hackers. *CACM*, 27(11):1085, November 1984.

[4] K. Thompson. Reflections on trusting trust. *CACM*, 27(8), August 1984.

[5] C. Young. Taxonomy of computer virus defense mechanisms. In *Proc. 10th National Computer Security Conference*, pages 220–225, Sep. 1987.